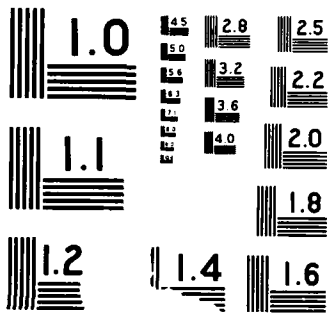
AD-A192 503 OBJECT-ORIENTED PROGRAMMING IN C++(U) NORTH CAROLINA 1/1
UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE
J M COGGINS N00014-86-K-0680

UNCLASSIFIED F/G 12/5 NL


1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

To appear in Proceedings of the IEEE Atlanta Software Technology Conference, Atlanta, GA, November 12-13, 1987.

Object-Oriented Programming in C++

1987

James M. Coggins
Computer Science Department
University of North Carolina
Chapel Hill

N00014-86-K-0680

Abstract

Object-Oriented design refers to a set of disciplines for organizing large software systems. Language support for object-oriented design includes facilities for data abstraction and inheritance. C++ is an extension of the C programming language that directly supports object-oriented constructs. Key features of C++ include support for classes, objects, and messages, operator overloading, and simplified heap storage manipulations. This paper describes how these features affect software development strategies. An idiom developed in our laboratory is illustrated in an example.

Introduction

Object-oriented Programming has been described as being "in the 1980's what 'structured programming' was in the 1970's" [1], and this analogy turns out to be useful as well as clever.

Structured programming is a coding discipline whose application yields low-level code structures that are easier for people to understand (and therefore easier to debug and maintain). Reasoned arguments for adopting this discipline involve measures of code complexity, the applicability of formal and informal verification methodologies, the theoretical power of various control constructs, and anecdotal evidence concerning the understandability of code. Additional (quite entertaining) religious debates have centered on the appropriate use of the **goto** construct and the value of various alternative control structures.

Object-oriented programming is a code packaging discipline that allows a designer to impose a reasonable structure on large software systems based on the notions of *encapsulation* and *inheritance*. This discipline provides an operational definition of *module*, an organizing principle for task decomposition, and a useful formal separation between architecture and implementation. Object-oriented design simplifies the structure of large software systems just as a decade ago structured programming simplified the structure of code segments. In addition, object-oriented design clarifies some system design responsibilities and gives us some new tools for thinking about software design.

It is possible, in principle, to adopt object-oriented design for systems written in languages that do not directly support object-oriented constructs. As with structured programming, however, there are great advantages in having a language embody and enforce the discipline. C++ provides support for the constructs of object-oriented programming while maintaining compatibility with C [2]. An example of C++ programming will be used to illustrate several object-oriented programming concepts.

Object-Oriented Programming

"Object-oriented" is rapidly becoming a high-tech synonym for "good". The meaning of "object-oriented programming" can be recovered by tracing its lineage in programming paradigms and the constructs supporting those paradigms. (This section is adapted from [3].)

Procedural programming was made possible by the development of closed subroutines. The ability to isolate name spaces and to explicitly indicate information transfers through argument lists permits a partial separation of concerns between implementers and users. The discipline of *data hiding* , supported by the technical innovation of object libraries, allows a partial separation of concerns between architects and implementers but provides little flexibility for users. *Data abstraction* extends the data hiding concept by providing support in the programming language for *abstract* or *user-defined data types.* With data abstraction, the organizing principle for developing large systems is *encapsulation;* one defines types so as to minimize the communication bandwidth among them (and among their implementation teams). The development of large systems involves deciding what types are needed and providing a full set of operations for each type. The abstract data type becomes the operational definition of a *module*, supporting a nearly complete formal separation of concerns between users, architects, and implemeters. The term *object-oriented programming* is often used as a synonym for data abstraction, but the term properly refers to the use of class hierarchies and inheritance to make explicit the commonalities among abstract data types. (See [1,3,4] for further discussion of the significance of object-oriented design.)

Encapsulation

When using object-oriented design, logically related data and operations are packaged together (encapsulated) in a *class.* An instance of a class is called an *object.* The class definition specifies both the memory structure and the set of allowed operations for objects of the class.

The operations and data in a class may be private, so only objects belonging to the same class may access them, or public, so objects of any class can use them. (C++ also supports an intermediate level of isolation called *protected*. Protected components may be accessed by objects of derived classes only.) The support for public and private components creates an explicit separation of the concerns of users, architects, and implementers. The public definition of a class is a contract between the users of the class and its architects that certain functionality will be supported. The set of class declarations consitutes a contract between the architect and the implementer that certain functionality and certain relationships among the classes will be supported. The implementation of the class structures is then the implementer's domain. This separation of concerns focusses the negotiations between users and architects on how the class structure can be defined to most faithfully reflect the user's mental model of the objects involved, which is exactly where their discussion should be centered.

The operations*(methods)* defined for a class are invoked on an object by sending a *message* to the object. The message names *what* is to be done by the object but not *how* it is to be done. "How" questions are reserved to the implementer. In particular, the implementer may write the code for the required function in the method, or the processing may be deferred to other objects by simply sending messages to request appropriate actions by other objects.

Constructors and destructors are special messages that are invoked automatically when an object is created (by declaration or by explicit allocation) or destroyed (by exiting the scope of the object or by explicit deallocation) to ensure that the object is initialized or deallocated correctly. The explicit inclusion of these essential operations gives the class definition a consistent and complete structure. In C++, constructors and destructors are explicitly defined in a uniform fashion (the constructor has the same name as the class; the destructor name is the class name preceeded by a tilde).

The analogy between classes and types and between objects and variables is obvious, but the object-oriented constructs are more than a renaming of familiar ideas. The significance of classes can be compared to the significance of the Pascal **record** or the C **struct**. Records and structs allow logically related data items of different types to be encapsulated together under a single name and then hierarchically organized. This encapsulation serves both to agglomerate related items and to separate

unrelated items. The language support for these constructs enforces the discipline defined in the record or struct declarations. Classes provide a similar organization for data and code together. Thus, the class structure organizes the software system and the data structures simultaneously.

To illustrate how data abstraction and encapsulation affect problem decomposition, consider the modularization of a simple compiler. The compiler will operate in three passes: lexical scan, parsing, and code generation. A decomposition of the compiler project that assigns one team to each pass imposes a heavy communication load between the teams. For example, the teams must jointly design the symbol table, token list, parse tree, and code list data structures. In an object-oriented design, these objects would be the principal modules, and each module would be written by one team. Negotiations between the teams then focus on the functionality and user interface of the modules; the internal structures and algorithms are hidden inside the modules. When the basic classes are designed, additional classes can be created for the lexical scanner, parser, and code generator. These *process encapsulations* focus on controlling interactions among the objects involved in each pass without the distracting details of internal representations of the basic objects. Objects that have as their purpose the control of interactions among other objects are called *enzymes* or *catalytic objects.*

The compiler example illustrates how data abstraction clarifies several practical issues in the management of large software projects. The distinction between public and private data and operations provides a useful separation of concerns between user interface, architecture and implementation. The class structure focuses discussion among users and architects on the public structure of the classes and how this public structure can best reflect the mental constructs they represent.

Inheritance

Object-oriented design permits class definitions to be hierarchically organized, leading to a powerful and subtle design tool called *inheritance.* With inheritance, the memory and message structures of a base class are inherited by all classes derived from the base class. (In object-oriented programming literature, the terms *base class* and *derived class* are synonyms of *superclass* and *subclass*, respectively.) Inheritance allows code to be shared between classes with similar structure. Variations of a class can be defined by specifying in the subclass definitions only those aspects of the subclasses that differ from the base class. Messages defined in the base class may be redefined in the derived class to handle

peculiarities of the derived class. The inheritance structure allows modifications to an implementation to be localized in the base class and automatically propagated to the derived classes.

Sometimes a class may be defined simply to hold structures common to several subclasses; no objects of this base class will ever be defined. Such a class is called an *abstract superclass*. Sometimes a superclass needs to be able to accept messages that must be interpreted in different ways by its subclasses. For example, if the subclasses store data of different primitive types, then different type casts will be required in each subclass. But in order for the superclass to receive the messages at all, the messages must be declared in the superclass. The *virtual function* mechanism in C++ allows the superclass to receive a message for which the interpretation must be supplied by a subclass. With virtual functions, an object may be known to be a `buffer` but whether it is an `int_buffer` or a `float_buffer` might be unknown at compile time. If `buffer` is the superclass with appropriate virtual functions defined, then the selection of the appropriate code to execute will be determined at run time based on the subclass of the object that is receiving the message.

Operator overloading

One of the most useful features of C++ in practice is operator overloading. With operator overloading, the names of messages need not be distinct, even in the same class. The appropriate message to execute is determined by the message name, the number of arguments, and the types of the arguments jointly. Furthermore, symbolic operators such as +, =, <<, +=, *=, etc. may be redefined to have valid interpretations for arguments of any class. Operator overloading has had two important uses in our lab. First, we can write code that is largely type-independent; we do not need different procedure names for performing the same operation on different types of objects. Second, we have made the use of matrices, vectors, and complex numbers much more natural by defining symbolic operators for the common operations; we do not need to remember, for example, whether matrix multiplication is mpy or mult since the operators * and *= are defined for class matrix.

Storage management

C++ provides operators **new** and **delete** for creating and deallocating objects in the heap. **New** returns a pointer to an object of the desired class located in the heap. When the object is allocated, a constructor is invoked to initialize the object. Similarly, **delete** invokes the destructor for an object's class and deallocates the object given a pointer to the object.

In our laboratory, we work with large objects such as images and graphical models. We have developed an idiom that provides type independence in writing user code, efficient storage management, and natural, direct methods for implementing our objects. The technique involves defining a *header class* or an *intelligent pointer class* that holds some administrative information about the object along with a pointer to an object belonging to a *storage class* that contains the mass of data. We can then manipulate the objects of the header class directly and naturally while the real activity is deferred to the storage class where the data is found. Type independence can be achieved if the storage class consists of an abstract superclass with subclasses for each primitive storage type. There are some rather subtle interactions between the semantics of C++ functions and the constructors and destructors of the header class, but when these are resolved the resulting code is elegant and efficient.

Example: Binary tree sort

The example on the next pages illustrates most of the features discussed above except inheritance, which would require a larger example. The program defines a binary tree node (btnode) having left and right subtree pointers and an integer data value. A private message, insert, is invoked by the overloaded public messages operator+(). Integers are inserted into the tree so that an infix traversal of the tree visits the data values in increasing order. This program is not concerned with balancing the sort tree. The messages return "self" to allow chaining of operations as illustrated in the main program. Recursive calls to print_infix and to operator+() are noted. Comments follow the double slashes.

```
// The following class definition would normally be stored in
// a header file such as btnode.h and would consititute part
// of the documentation of the class.

class btnode{
   btnode* left,right;          //  Private data comes first
   int  data;
   btnode insert(int);            //  note: a private message
public:                          //  User interface routines follow
   btnode();                      //  null constructor
   btnode(int);                   //  another constructor
   ~btnode();                     //  destructor
   btnode print_infix();
   btnode operator+(int);         //  These messages illustrate
   btnode operator+(btnode);      //  operator overloading
}
```

```
//  The following implementations would normally be stored in a
//  C++ file such as btnode.c and would not normally be examined
//  by users of the btnode class.

btnode::btnode()
{                       // the null constructor initializes the
   left=NULL;           // object to all default values
   right=NULL;
   data=0;
}

btnode::btnode(int i)
{                       //  this constructor initializes the node
   left=NULL;           //  with a particular data value
   right=NULL;
   data=i;
}

btnode::~btnode()          //  the destructor just has to be sure
{                          //  that the subtrees get deallocated
   if(left<>NULL) delete left;       // Note: the destructor will
   if(right<>NULL) delete right;     // be invoked by each node in
}                                    // the subtree as it is deleted

btnode btnode::insert(int i)
{
   if (i<data) then
      if (left=NULL) then
         left=new btnode(i);  // allocate a new btnode
      else
         left->insert(i);     // left is a pointer; use -> syntax
   else if (i>data) then
      if (right=NULL) then
         right=new btnode(i); // allocate a new btnode
      else
         right->insert(i);    // send to right subtree
   return self;
}

btnode btnode::print_infix()
{
   // This message uses recursive calls to itself to print the
   //   tree in infix (sorted) order
   if(left<>NULL) left->print_infix();   // print my left subtree
   cout << data << " ";                  // print my data
   if(right<>NULL) right->print_infix(); // print my right subtree
   return self;
}

btnode btnode::operator+(int i)
{                                 // add an integer to the sort tree
   self.insert(i);     // (note: I'm sending a message to myself!)
   return self;        //  since self is an object, use . syntax
}
```

```
btnode btnode::operator+(btnode b)
{                                   // add a tree into self!!
   if(b.left<>NULL)then
      self+(*b.left);               // * dereferences the pointer
   if(b.right<>NULL) then           //  note the recursive calls to +
      self+(*b.right);
   self+b.data;       //  This one is NOT recursive!
   return self;
}

// The main program would normally be stored in its own file,
// say main.c, and would need to #include the btnode.h file
// above as well as some system include files.  The btnode.c
// file would be compiled into an object library and linked
// in at run time.

main()
{
   btnode s,t;
   s+6+3+2+5+8+7;              // chaining is possible because of
   t+4+9+1+10;                 // the return self; statements
   s.print_infix(); cout << "\n";
   t.print_infix(); cout << "\n";
   s+t;                        // tree plus tree gives tree!
   s.print_infix(); cout << "\n";
}
0 2 3 5 6 7 8
0 1 4 9 10
0 1 2 3 4 5 6 7 8 9 10
```

Acknowledgement

This research was supported in part by ONR contract N00014-86-K-0680.

References

1. Rentsch, Tim, "Object-Oriented Programming", *SIGPLAN Notices*, vol. 17, no. 9, September 1982, pp. 51-57.

2. Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.

3. Stroustrup, Bjarne, "What is Object-Oriented Programming?", *Proc. 1st European Conference on Object-Oriented Programming*, Paris, 1987. In press in *Lecture Notes in Computer Science,* vol. 276, Springer Verlag.

4. Cox, Brad, *Object-Oriented Programming: An Evolutionary Approach,* Addison-Wesley, 1986.

END

DATE

FILMED

6-88

DTIC